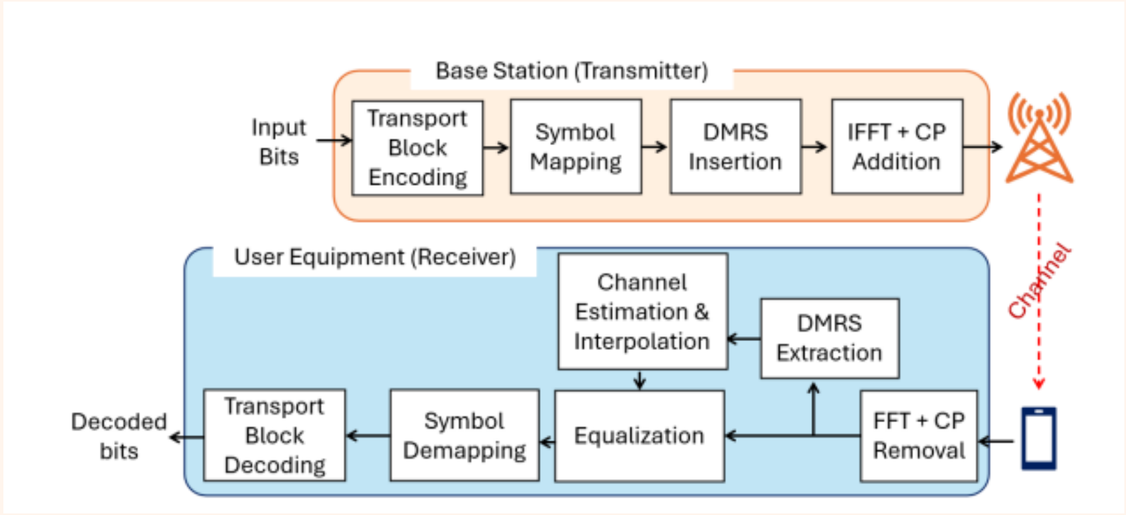
# SpikingRx

B11102233 林孝陽

# 大綱

1. 傳統5G發送接收
2. SpikingRx架構
3. surrogate gradient
4. 系統效能

# 傳統5G發射端




Source

01 LDPC Encoder

02 Modulator

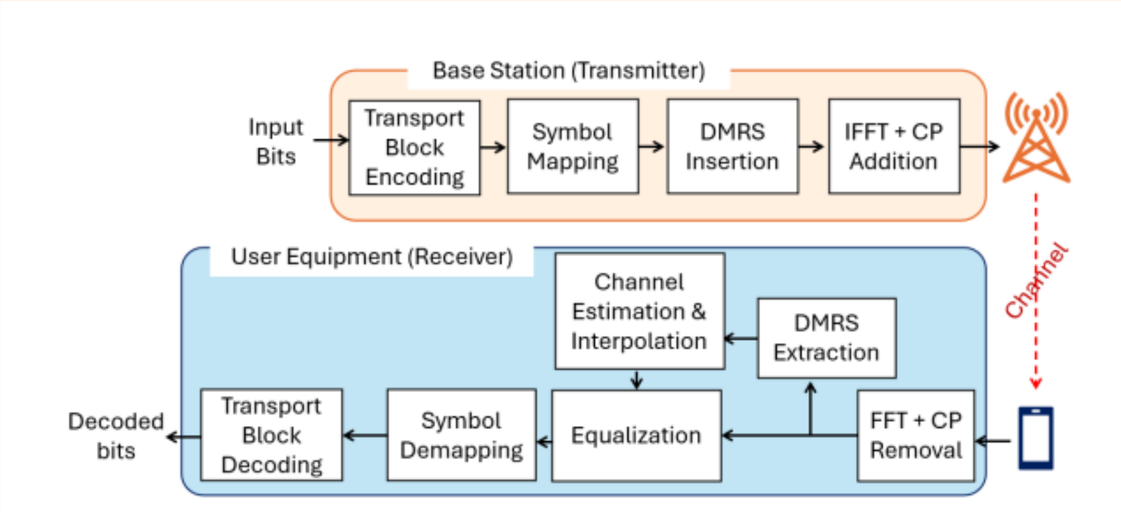
03 DMRS

04 IFFT

05 CP

Channel

# 傳統5G接收端




Channel

05 FFT+cp-removed

06 Channel Estimation

07 Equalizer

08 Demapper

09 LDPC Decoder

# LDPC Encoder

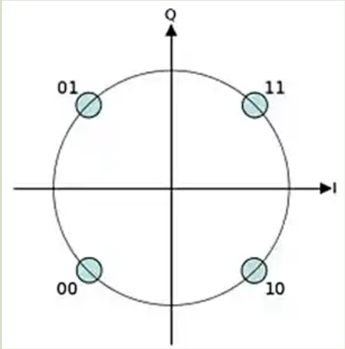
利用5G NR的BG基圖裡子矩陣循環位移的關係，對資訊位進行 XOR 運算以生成 parity bits

→

```python
# === 假設 info bits（k=8*2=16 bits）===
info = np.random.randint(0, 2, Z * 2)  # 2 個資訊區塊
print("Info bits:", info)

# === 生成 parity bits ===
parity = np.zeros(Z, dtype=int)

# 以第一個 parity 區塊為例
# parity = XOR(右移後的資訊區塊們)
for i, shift in enumerate([0, 1]):
    if shift != -1:
        shifted = np.roll(info[i*Z:(i+1)*Z], shift)
        parity ^= shifted  # XOR

print("Parity bits:", parity)

# === 組合成 codeword ===
codeword = np.concatenate([info, parity])
print("Codeword:", codeword)
```

# Modulator

QPSK將 codeword 依照 2 bits 一組映射到星座圖（ Gray 編碼 ）




```
def qpsk_mod(bits):
    bits = bits.reshape(-1,2)
    mapping = {
        (0,0): 1+1j,
        (0,1): -1+1j,
        (1,0): 1-1j,
        (1,1): -1-1j
    }
    return np.array([mapping[tuple(b)] for b in bits])

symbols = qpsk_mod(codeword[:8])  # 假設只取前8 bits
print("QPSK symbols:", symbols)
```

# DMRS

將QPSK符號映射到 OFDM
子載波 子載波彼此正交

特定子載波與時槽插入
DMRS，讓接收端能估計通道

→

```python
# 假設有 4 個子載波，index 1 放 DMRS
num_subcarriers = 4
dmrs_index = 1
dmrs_symbol = 1 + 1j  # 簡單示意 DMRS

# 預留符號空間
ofdm_freq = np.zeros(num_subcarriers, dtype=complex)

# 插入資料符號
data_indices = [i for i in range(num_subcarriers) if i != dmrs_index]
ofdm_freq[data_indices] = symbols[:len(data_indices)]

# 插入 DMRS
ofdm_freq[dmrs_index] = dmrs_symbol

print("OFDM freq-domain with DMRS:", ofdm_freq)
```

# IFFT

將頻域符號 X[k] 轉成時域訊號 x[n] 為了RF傳輸

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j2\pi kn/N}$$

```
ofdm_time = ifft(ofdm_freq)
print("OFDM time-domain signal:", ofdm_time)
```

# CP

把最後幾位元加到最前面

對抗多徑衰落與 ISI

```
cp_len = 1  # 簡單示意
ofdm_cp = np.concatenate([ofdm_symbols[-cp_len:], ofdm_symbols])
print("OFDM with CP:", ofdm_cp)
```

# Channel

假設真實通道 H_real多路徑

w[n]為AWGN

$$y[n] = \sum_{l=0}^{L_h-1} h[l]x_{\text{cp}}[n-l] + w[n]$$

```
# 假設真實信道 H_real
H_real = np.array([1.0+0.0j, 0.8+0.2j, 1.1-0.1j, 0.9+0.1j])
rx_ofdm = ofdm_time * H_real + 0.05*(np.random.randn(num_subcarriers) + 1j*np.random.randn(num_s
```

# CP removed+ FFT

接收端移除前面幾位元的cp

對 r[n] 做 FFT 得到頻域符號 Y[k]

$$Y[k] = \sum_{n=0}^{N-1} y_{\text{cp\_removed}}[n] e^{-j2\pi kn/N}$$

→

```python
rx_no_cp = rx_signal[cp_len:]  # 移除 CP
rx_symbols = np.fft.fft(rx_no_cp)
print("Received symbols:", rx_symbols)
```

# Channel Estimation

利用接收的DRMS跟原始DRMS比較得到估計信道H

$$\hat{H}_{\mathrm{LS}}[k] = \frac{Y_{\mathrm{DMRS}}[k]}{X_{\mathrm{DMRS}}[k]}$$

→

```python
# 估計整個子載波信道，這裡用最簡單方法：DMRS 插值
H_est = np.zeros(num_subcarriers, dtype=complex)

# 先用 DMRS 估計該子載波
H_est[dmrs_index] = rx_freq[dmrs_index] / dmrs_symbol

# 簡單線性插值估計其他子載波信道
for i in data_indices:
    # linear interpolation between DMRS and adjacent (簡單示意)
    H_est[i] = H_est[dmrs_index]

print("Estimated channel H_est:", H_est)
```

# Equalization

再用估計的 H 做等化回資料符號

ZF: $\tilde{X}[k] = Y[k] / \hat{H}[k]$

LMMSE: $\tilde{X}[k] = \frac{\hat{H}^*[k]}{|\hat{H}[k]|^2 + \sigma^2} Y[k]$

→

```
eq_symbols = rx_freq[data_indices] / H_est[data_indices]
print("Equalized symbols:", eq_symbols)
```

# Soft Demodulation

把複數頻域符號拆成 bit 的軟資訊（LLR），送 LDPC 解碼

$$LLR_{\text{real}} = \frac{2}{\sigma^2}\,\text{Re}\{\tilde{X}[k]\}, \quad LLR_{\text{imag}} = \frac{2}{\sigma^2}\,\text{Im}\{\tilde{X}[k]\}$$

→

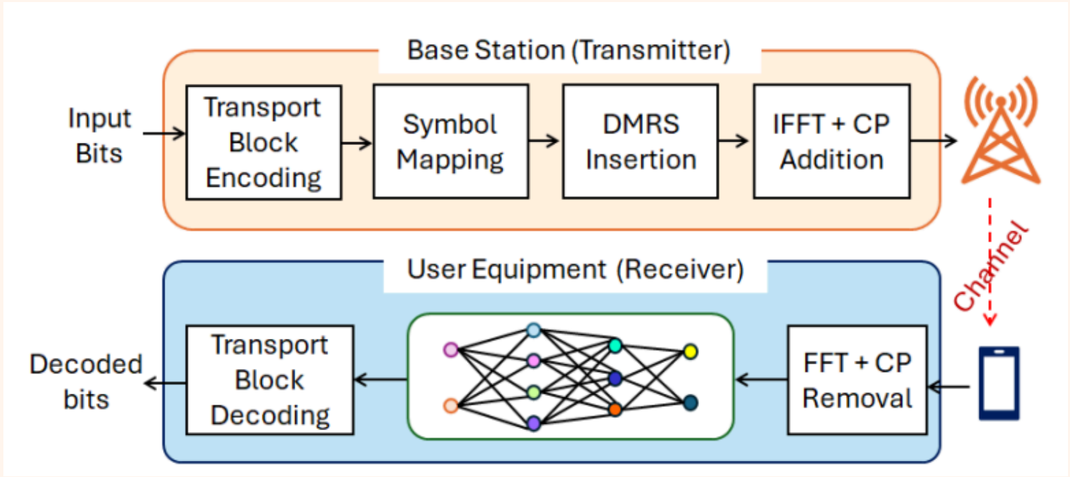
```python
def qpsk_llr(symbols):
    llr = []
    for s in symbols:
        llr.append(np.real(s))  # 簡單示意: real -> bit0, imag -> bit1
        llr.append(np.imag(s))
    return np.array(llr)

llr_bits = qpsk_llr(eq_symbols)
print("LLR bits:", llr_bits)
```

# LDPC Decoder

使用 sum-product 或
min-sum 演算法
利用 LLR 計算每個 bit 的
posterior probability
→ 還原原始 bits

```
decoded_bits = (llr_bits > 0).astype(int)
print("Decoded bits:", decoded_bits[:len(data_bits)])
```

# SpikinrRx架構




Channel

05 FFT+cp-removed

06 SpikingRx

07 SpikingRx

08 SpikingRx

09 LDPC Decoder

# CP removed+ FFT

接收端移除前面幾位元的cp

對 r[n] 做 FFT 得到頻域符號 Y[k]

$$Y[k] = \sum_{n=0}^{N-1} y_{\text{cp\_removed}}[n] e^{-j2\pi kn/N}$$

→

```python
rx_no_cp = rx_signal[cp_len:]  # 移除 CP
rx_symbols = np.fft.fft(rx_no_cp)
print("Received symbols:", rx_symbols)
```

# 前處理

將複數符號拆成實部與虛部
→ 形成神經元輸入

對每個神經元輸入做 rate coding → 將其映射成 spike trains(週期)
振幅越大 spike次數越多

→

```python
import numpy as np

# === 1. 假設 FFT 後複數符號 ===
Y = np.array([0.7 - 0.2j, -0.5 + 0.8j])
real = np.real(Y)
imag = np.imag(Y)

# === 2. 拆實部與虛部，形成輸入向量 ===
x_in = np.concatenate([real, imag])
print("Input vector (real + imag):", x_in)

# === 3. Rate coding (振幅越大 spike 越頻繁) ===
# 這裡簡單用隨機模擬 spike train，10 個時間步
rate = np.abs(x_in) / np.max(np.abs(x_in))
spike_train = np.random.rand(10, len(x_in)) < rate
print("Spike train (10 time steps):")
print(spike_train.astype(int))
```

# 初始化權重W

卷積核權重初始化常用

Kaiming Normal

最開始每個神經元通道的權重

服從這個分布

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right)$$

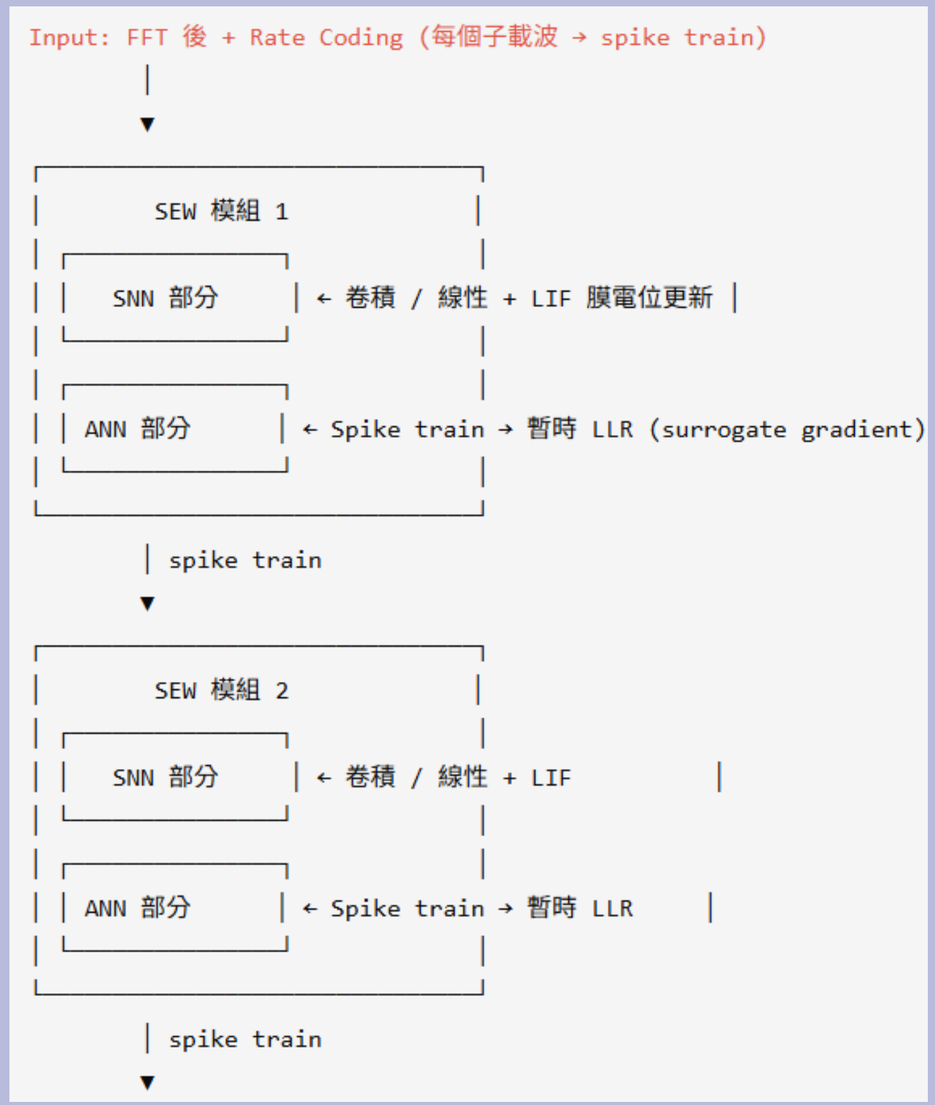
→

```python
# === 4. SNN 權重初始化 (Kaiming Normal) ===
n_in = x_in.shape[0]
n_out = 4  # 假設 4 個神經元
W = np.random.normal(0, np.sqrt(2/n_in), size=(n_out, n_in))
print("Initialized weights W:")
print(W)
```

# SEW-ResNet

Block 結構

- 每個 Block 由 多層 SNN + 最後一層 ANN 組成。
- SNN 部分：
  - 每層的每個通道（channel）同時計算，累積膜電位並放電。
  - 正向傳播產生脈衝序列（spike trains）。
- ANN 部分：
  - 將 SNN 脈衝輸出整合
  - 計算最終 LLR（soft bits）
  - 透過 LLR 計算 loss，並利用 surrogate gradient 更新不同通道自己的權重 W



# SNN層(單通道)

- 每個神經元通道會接收一串脈衝序列作為輸入
- 在每個時間步：
  a. 將輸入脈衝乘以對應權重W，加到累積膜電位上。
  b. 檢查膜電位是否達到放電閾值
     - 若達到 → 神經元放電，輸出 1，並重置膜電位。
     - 若未達到 → 不放電，輸出 0，膜電位會隨時間自然衰減。
  c. 更新後的膜電位帶入下一個時間步，與新的脈衝累加。
- 這個過程對每個時間點循環進行，最終生成輸出脈衝序列

$$V[t+1] = \alpha V[t] + \sum_j W_{ij} S_j[t] - V_{\text{th}} \cdot S_i[t]$$

```
# 參數
num_neurons = 4       # 神經元數量
time_steps = 10       # 時間步數
Vth = 1.0             # 閾值
alpha = 0.9           # 膜電位衰減係數

# 初始化
V = np.zeros(num_neurons)                # 每個神經元膜電位
S = np.zeros((time_steps, num_neurons)) # 脈衝輸出序列
W = np.random.rand(num_neurons, num_neurons)  # 隨機權重

# 假設輸入脈衝序列
input_spike = np.random.randint(0, 2, size=(time_steps, num_neurons))

for t in range(time_steps):
    for i in range(num_neurons):
        # 計算膜電位
        V[i] = alpha * V[i] + np.dot(W[i], input_spike[t]) - Vth * S[t-1, i] if t>0 else alpha*V[i

        # 檢查是否放電
        if V[i] >= Vth:
            S[t, i] = 1
            V[i] = 0  # 放電後重置
        else:
            S[t, i] = 0

print("脈衝輸出序列 S:\n", S)
```

# ANN 層與迭代流程

- 脈衝整合與 LLR 推估
  - 將每個通道輸出的脈衝序列在整個時間軸上，累加脈衝次數得到對應的 LLR ( soft bits )
  - 用簡單比例：脈衝數越多 → 越傾向 1
- Loss 計算與梯度更新
  - 將每個神經元輸出的 LLR 與原始 bit做比較
  - 用MSE計算 loss
  - 使用三角形替代梯度 ( surrogate gradient ) 更新權重 W
- SEW 模組迭代
  - 將SNN計算出的脈衝序列再輸入一次 SEW
  - 使用更新後的權重 W
  - 反覆多次，直到 loss 收斂 → LLR 也收斂

MSE loss :

$$L = \frac{1}{N}\sum_{i=1}^{N}(\mathrm{LLR}_i - b_i)^2$$

```python
def compute_loss(LLR, target_bits):
    # 將 LLR 視為 soft bits
    return np.mean((LLR - target_bits)**2)
```

# Surrogate Gradient

對某個連結權重 W，用鏈式法則計算梯度

- L：損失（loss），衡量網路輸出與目標差距
- S：神經元輸出脈衝（0 或 1），forward 時是 step，backward 用 surrogate gradient
- V：膜電位
- W：要更新的權重
- η：學習量

σ′(V−Vth) → surrogate gradient(本文使用三角形)

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial S} \cdot \frac{\partial S}{\partial V} \cdot \frac{\partial V}{\partial W}$$

$$\Delta W = -\eta \cdot \frac{\partial L}{\partial W} = -\eta \cdot (\frac{\partial L}{\partial S}) \cdot \sigma'(V - V_{th}) \cdot \frac{\partial V}{\partial W}$$

# ∂L/∂LS=g

- 損失對神經元輸出脈衝的敏感度
- 計算方式：由最終 loss + 後層影響 chain rule 傳回
- 若 LLR 是多個 spike 則輸出線性組合

$$L = \frac{1}{2}(LLR_{pred} - LLR_{target})^2$$

$$\frac{\partial L}{\partial LLR_{pred}} = LLR_{pred} - LLR_{target}$$

$$LLR = \alpha \cdot spike\_count + b$$

$$\frac{\partial L}{\partial S} = \frac{\partial L}{\partial LLR_{pred}} \cdot \frac{\partial LLR_{pred}}{\partial S} = g$$

# ∂S/∂V≈σ′(V−Vth)

- 取代不可偏微的脈衝函數
- 三角形 surrogate gradient ：σ′(x)
- a：控制梯度寬度
- 功能：即使神經元當下沒放 spike，只要膜電位接近閾值，也能得到非零梯度
- 越靠近閥值， σ′(V−Vth)越大

$$\sigma'(x) = \max(0, 1 - |x|/a)$$

# ∂V/∂W

- 膜電位對權重的偏微
- 對特定權重 Wk輸入脈衝為 1 → 對應梯度 = 1；輸入為 0 → 對應梯度 = 0

$$V[t] = V[t-1] + \sum_j W_j S_j^{in}[t]$$

$$\frac{\partial V}{\partial W_k} = S_k^{in}[t]$$

# EX:

| 參數 | 數值 |
| --- | --- |
| V | 0.8 |
| Vth | 1.0 |
| a | 1.0 |
| x = S_in | 1 |
| pred | 0.2 |
| target | 1.0 |
| α (LLR→脈衝係數) | 0.625 |
| η | 0.01 |

1. g = ∂L/∂S

$$g = (pred - target) \cdot \alpha = (0.2 - 1.0) \cdot 0.625 = -0.5$$

2. surrogate gradient

$$\sigma'(V - Vth) = 1 - |0.8 - 1|/1 = 0.8$$

3. 膜電位對權重

$$\frac{\partial V}{\partial W} = x = 1$$

4. 梯度

$$\frac{\partial L}{\partial W} = g \cdot \sigma' \cdot x = -0.5 \cdot 0.8 \cdot 1 = -0.4$$

5. 權重更新

$$\Delta W = -\eta \cdot \frac{\partial L}{\partial W} = -0.01 \cdot (-0.4) = +0.004$$

6. 新權重

$$W_{new} = W_{old} + \Delta W = W_{old} + 0.004$$

# 系統效能

**A** 錯誤率指標（Error Performance）

**B** 神經元激活與能量消耗（Neuron Activation & Energy Efficiency）

**C** 系統穩健性（Robustness）

# 錯誤率指標

- BLER：用來評估接收端對 OFDM 資源格的解碼正確性。
- 對比基準：
  - NeuralRx：ANN-based 接收器。
  - 5G-NR LS Estimation：傳統接收器（線性最小平方估計 + LMMSE 等化器）。
  - Perfect CSI：理想接收器，知曉完美信道資訊，作為理論下界。

# 錯誤率表現

- 與 NeuralRx 相近，1~2 DMRS 符號下接近 Perfect CSI。BLER 與 NeuralRx 相差僅 0.2 dB
- 結論：SpikingRx 的錯誤率表現幾乎與 ANN-based NeuralRx 相當，且優於傳統線性接收器。

# 神經元激活與能量消耗

- 激活概率：神經元在不同時間步產生 spike 的比例。
- FLOPS / Energy Consumption：依據 spike 發生與否計算 SNN 與 ANN 的運算量與能量成本。
- 量化後的 Quantized SpikingRx：用低精度權重進一步降低能量消耗。

# 神經元激活與能量效率

- 激活概率：
  - 第一層神經元激活率最低，之後相對均勻 → 信息逐層精煉。
- 能量消耗：
  - 與 NeuralRx 比較：
    - T=2 → 約 8.9× 節能。
    - T=10 → 約 2.1× 節能。
    - Quantized SpikingRx → 可再降低 5×。
- 結論：SpikingRx 在保持相似性能下，能大幅降低運算能量。

# 系統穩健性

- 量化穩健性：8-bit 或 32-bit 權重對性能的影響。
- 通道條件變化的魯棒性：不同 SNR、Doppler、LOS/NLOS 情況下的 BLER 表現。
- 速度變化（UE Speed）：低速/中速/高速場景對解碼效果影響。

# 系統穩健性

- 對量化影響小：8-bit 權重僅比 32-bit 權重差 0.1 dB BLER。
- 對通道變化敏感度低：不同 SNR、Doppler、LOS/NLOS 下仍保持穩定解碼。
- 對 UE 速度敏感度低：從 3.6 km/h 到 108 km/h，BLER 變化很小。
- 結論：SpikingRx 對硬體量化與實際通道波動具有高度魯棒性。